Armada: A Robust Latency-Sensitive Edge Cloud in Heterogeneous Edge-Dense Environments

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Lei Huang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Abhishek Chandra, Jon Weissman

July, 2021

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school. I would like to thank my advisors, Dr. Abhishek Chandra and Dr. Jon B. Weissman, for their help and guidance that has seen me through this thesis. I would like to express my gratitude to the other member of my examination committee, Dr. Georgios Giannakis, for his patience and support. I would also like to thank the entire computer science faculty and all my fellow graduate students for their support. Especially thanks to Nikhil Sreekumar, Zhiying Liang, Cody Perakslis and Sumanth Kaushik Vishwanath.

Specifically, I would like to acknowledge Zhiying Liang's contributions in implementing the service scheduler (3.4.1) and Nikhil Sreekumar's contributions in design and development of the storage layer (3.5). I would also like to thank Cody Perakslis and Sumanth Kaushik Vishwanath's help with the system development and testing.

Abstract

Edge computing has enabled a large set of emerging edge applications by exploiting data proximity and offloading latency-sensitive and computation-intensive workloads to nearby edge servers. However, supporting edge application users at scale in wide-area environments poses challenges due to limited point-of-presence edge sites and constrained elasticity. In this paper, we introduce Armada: a densely-distributed edge cloud infrastructure that explores the use of dedicated and volunteer resources to serve geo-distributed users in heterogeneous environments. We describe the lightweight Armada architecture and optimization techniques including performance-aware edge selection, auto-scaling and load balancing on the edge, fault tolerance, and in-situ data access. We evaluate Armada in both real-world volunteer environments and emulated platforms to show how common edge applications, namely real-time object detection and face recognition, can be easily deployed on Armada serving distributed users at scale with low latency.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Edge computing, a computing paradigm that brings computation closer to data sources and end-users, has enabled the deployment of emerging edge-native applications [1, 2]. With 5G accelerating the first network hop and rapid rollout of public edge infrastructure, edge computing is starting to play a significant role in the computing landscape [3].

The emerging edge-native applications, including AR/VR, cognitive assistance, autonomous vehicles, are latency-sensitive and compute-intensive. Offloading workload from devices to powerful edge servers that can run complex machine learning algorithms is necessary to resolve the device-side limitation. The demand for these applications will increase rapidly and require the edge to be highly available and scalable. However, elasticity is a well-known limitation of edge resources [4]. A burst of incoming workload can easily overwhelm an edge site causing service performance degradation. Furthermore, widely geo-distributed users require wide edge availability with full coverage of geographical locations to provide low-latency edge access. These requirements cannot be satisfied by single providers with limited point-of-presence and capacity in today's edge infrastructure deployments [5, 6, 7, 8].

1.2 Challenges and our solution

Edge platforms that exploit edge resources from multiple providers have been proposed in both industry [9, 10, 11], and academia [12] to enlarge the edge coverage. However, they are built on top of dedicated resources with a *sparsely-distributed resource model*: users from a certain geographic location only have one or few nearby edge options which can provide a low-latency response. Overload can easily happen since dedicated resources are physically limited and lack scaling capabilities. With the advent of powerful personal computers and devices, we believe the necessary compute power is already closer to the users. Volunteer-based underused personal devices can be organized and coordinated at scale to resolve resource limitations on the edge. In this thesis, we introduce Armada, a robust latency-sensitive edge cloud that explores the use of both dedicated and volunteer resources to support low-latency computation offloading.

Armada uses a *densely-distributed resource model*: users from a certain geographic location can have multiple nearby options to offload computations. Specifically, we explore the following challenges:

- How to select edge nodes to obtain low end-to-end latency in heterogeneous environments?
- How to achieve edge scalability with multiple loosely-coupled and resource-constrained edge nodes?
- How to guarantee continuous service in volunteer environments with high node churn and failure rate?
- How to minimize latency overhead for data persistence and consistency on edge?

Armada implements auto-scaling service deployment mechanisms based on real-time user demand and distribution, and uses a user-side performance probing strategy as a key idea to guide service selection and load balancing among multiple edge nodes. The service deployment mechanisms incorporate several factors that affect performance, including user/data geo-location, edge server load, and network latency. User-side probing employs multiple, flexibly maintained client-to-edge connections that provide fault tolerance by enabling immediate connection switch to alternate edge nodes upon node

failure. In addition, we introduce an edge-native storage layer to support low-latency data access when data and processing states cannot persist locally on volatile compute resources.

In this thesis, we focus on the system and implementation aspects of Armada. We show how real-time inference, a common latency-sensitive and computation-intensive application category, can be easily deployed on Armada and serve geo-distributed users with low latency. Then we take a closer look at system scalability, fault tolerance and data access performance in both real-world volunteer environments and emulation environments. The evaluation shows that Armada achieves a 33% - 52% reduction in average user end-to-end latency with high concurrent demand compared to locality-based and dedicated-resources-only approaches.

1.3 Contribution

We list our contributions as follows:

- Deliver low-latency edge service access through a performance-aware service selection approach.
- Achieve edge scalability through a densely-distributed edge resource model.
- Achieve fault tolerance on the edge through a redundancy-based multi-connection strategy.
- Implement an native storage layer on the edge to support in-situ data access and persistence.

1.4 Thesis Overview

- Chapter 2 briefly introduces the paradigm of edge computing and gives an overview of Armada driven by our design principles. It also includes a brief introduction of related work.
- Chapter 3 describes Armada system components in detail.

- Chapter 4 and 5 discuss two core strategies and building blocks Armada applies to achieve low-latency service selection, edge scalability and fault tolerance.
- Chapter 6 showcases the Armada workflow through two example real-time inference applications.
- Chapter 7 shows the experimental results that validate our solution.
- Chapter 8 gives the conclusion and discusses the future work.

Chapter 2

Background and Armada Overview

2.1 Edge Computing Paradigm

Edge computing is a computing paradigm that brings the compute resources physically closer to data generation, applications and end users to minimize the latency overhead over the networking communication channels. Instead of transferring data all the way to remote data centers for processing, the presence of edge resources enables low-latency responses for service and data access. It has become significant for emerging latency-sensitive and computational-intensive applications.

The origin of edge computing reaches back to the 1990s when the content delivery network (CDN) was introduced by Akamai [13] to accelerate web-based content delivery. By placing the caching and storage capacity geographically closer to end users, image and video data can be fetched faster to improve the web performance.

With the paradigm of cloud computing starting to thrive in 2006 [14], the concept of edge computing was first introduced by Satyanarayanan et al. in 2009 [15]. Edge computing generalizes the idea of CDN by bringing the compute capacity and cloud technology to the network edge. It can support highly responsive cloud services by minimizing the communication latency through short networking paths. With the number of Internet of Things (IoT) devices rapidly increasing, the term fog computing was proposed in 2012 [16] to cater the need of IoT applications. Besides the latency benefits

edge resources can provide, fog computing nodes concentrate on processing, filtering and aggregating data streams from IoT sensors and actuators locally to save the bandwidth consumption of remote centralized services and overall networking cost.

In the following years, the paradigm of edge computing [3, 17, 18] has become a significant concept with emerging latency-sensitive and AI-powered applications, including wearable cognitive assistance [19], video analytics for drones [20, 21], autonomous vehicles [22], AR/VR gaming [23] and real-time surveillance [24, 25]. Geographically-distributed edge computing infrastructure successfully extends the cloud data centers and brings profound impact to the future of computing landscape.

2.2 Armada Overview

2.2.1 Heterogeneous Edge-Dense Environment

Logical proximity, defined as low-latency high-bandwidth communication channels between edge servers and users, is usually provided by a LAN, on-premise networking infrastructures, and increasingly 5G technologies. However, special-purpose networking and compute resources on the edge are highly constrained in availability and scalability. In Figure 2.1, we show that nearby general-purpose resources in heterogeneous WAN environments (Edge-tier-2) can also provide low-latency benefits when Edge-tier-1 resources are not available or overloaded. We include both dedicated local public servers and volatile volunteer resources in Edge-tier-2 to enlarge the edge presence. Therefore, the resource limitation on edge can be resolved with the help of abundant volunteer edge nodes densely distributed around users, namely *edge-dense* environments.

The heterogeneity of Edge-tier-2 resources is twofold. First, connections from users to edge servers in WAN environments are highly diverse in terms of local ISPs and underlying networking infrastructure. Based on how users connect to the network, the actual number of routing hops and latency performance to the same edge server can highly diverge. Second, accessible compute resources present in nearby areas come from multiple providers and individuals. The heterogeneous capacity and hardware can lead to different processing performance, which is on the critical path of user requests and thus affects the end-to-end latency. Volunteer resources will amplify such heterogeneity by introducing more edge access points and increasing the system entropy.

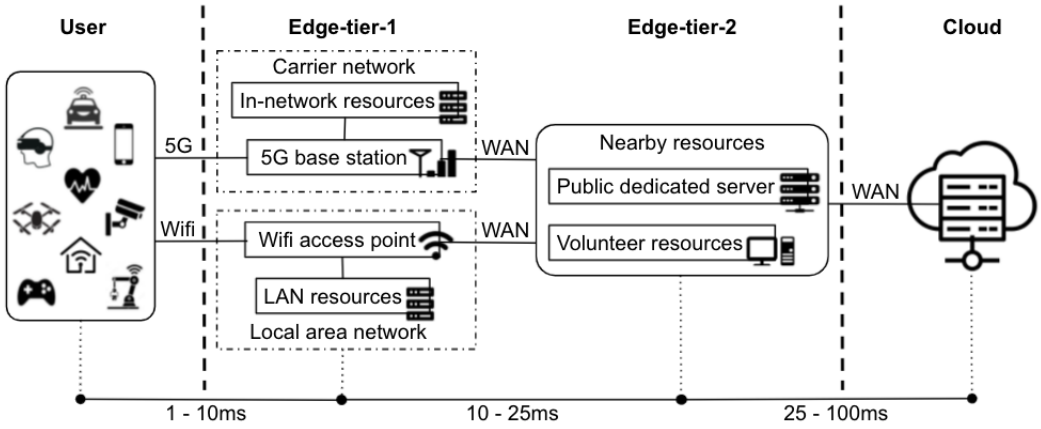

Figure 2.1: RTT latency in heterogeneous edge-dense environment

2.2.2 Design Goals

Armada is designed with the following goals in mind:

- **Support for low-latency computation offloading at scale with densely distributed edge resources:** While one edge server is limited by its capacity, many loosely coupled but densely distributed edge nodes can coordinate with each other to provision nearby users at scale. Armada is designed to manage resource-constrained but abundantly distributed edge nodes to support scalable low-latency computation offloading. As a result, applications deployed on Armada are able to automatically scale and obtain more resources in a specific region if more users are present.

- **Locality-based service deployment:** Service deployment should be based on fine-grained geographical specifications to reduce networking latency. Multiple replicas [1] of the service should be deployed on different edge nodes to guarantee edge availability and capacity in specified regions. Changes to currently active users should also dynamically guide the service placement to fit the real-time user distribution. Furthermore, new service deployment should be optimized for short startup time to start serving users in a timely manner.

[1] We use the term service replica and task interchangeably in this thesis.

- **Performance-aware service selection in heterogeneous environments:** Geographical proximity is not strictly equivalent to low RTT latency. Multiple factors together determine the edge performance including network/compute resource heterogeneity and availability. Given a list of nearby edge nodes running replicas of the application service, Armada should identify the best-performing edge access point for each user to offload the computation. This edge selection process should also handle the load balancing for all users to achieve overall lower latency.

- **Ease of use:** Armada interfaces should be easy to use for both application developers and resource contributors. In particular, developers should use Armada SDK with minimum code modifications to their applications for deployment. Moreover, resource contributors should be able to register their nodes quickly with lightweight components and isolated runtime.

- **Fault tolerance:** Armada must ensure the fault tolerance for Armada users in the presence of high node churn due to volatile, unreliable and unpredictable volunteer resources. Armada users must be guaranteed continuous service and experience zero downtime upon node failure or node leaving.

- **In-situ edge storage:** Armada should provide a native storage layer on the edge [26] to support low-latency data access. The storage layer should be reliable and independent from the volatile compute layer to persist the data for stateful and data-intensive applications. Also, flexible duplication and consistency policies should be supported for different application requirements.

2.2.3 Armada Application

Armada applications are long-running edge services using Armada resources for low-latency computation offloading. It includes a server-side program submitted to Armada for application-specific processing, and a client-side program used by application users to discover the service and offload computations. Armada deploys multiple replicas of the server-side program (tasks) to guarantee availability and scalability. Moreover, the client-side program uses Armada SDK to help application users locate the nearby

service access points and establish direct communication channels. In Armada, we focus on the scenario where application users are co-located with the processing data, such as AR users sending out video streams for real-time processing. However, we also support external data upload from other data sources to the Armada storage layer, providing low-latency data access for running services.

In Armada, volunteer resources are assumed to be unstable, volatile, and dynamic, with high node churn in heterogeneous environments. The guarantee on immediate recovery and continuous services upon node failure requires that application clients immediately switch connections to other service replicas and continue processing without waiting for failed node recovery. Therefore, no hard states or dependencies of the users are allowed to be maintained on the server-side for Armada applications. Application developers should either modify the application to maintain hard states and execution contexts on the client-side or use the Armada storage layer through Armada storage SDK to persist the data with minimized latency overhead.

2.3 Related Work

Edge sites are abundant in point-of-presence but limited in capacity. [4] demonstrates how an edge site can be easily overloaded with a burst of incoming traffic. Adaptive streaming and greedy-based optimization techniques are proposed to maximize the resource utilization per edge site, however, the client-centric QoS reception is not considered to make resource allocation decisions. [27] also focuses on improving the resource utilization from the perspective of a single edge site to achieve edge elasticity, but it fails to interpret the scalability problem in multi-edge environments. [28] considers load balancing over multiple edge sites for elasticity, but it is built on dedicated-only edge resources with limited capacity and flexibility. In dynamic volatile environments, [29, 30] utilizes client-centric metrics (QoS) to guide edge selection decisions, but they fail to consider the heterogeneous edge compute capacity and their effects on latency performance. [31] uses networked mobile devices as an extended cloud. It can tolerate edge server failures by enabling isolated device clusters to function separately, but it fails to consider the networking performance and resource heterogeneity of each mobile

device to identify the QoS. Also, the related works above are evaluated based on emulation platforms, lacking real-world experiments in edge environments. In Armada, we consider latency-sensitive edge selection, scalability and fault tolerance challenges all together to design the system architecture and mechanisms.

Several different research projects investigate the utilization of volunteer resources for both compute and storage [32, 33, 34]. Nebula [35] is a geo-distributed edge cloud that uses volunteer on an otherwise dedicated resource system to carry out data-intensive computing infrastructure for intensive computation and data storage with a NaCI sandbox. The NaCl sandbox is limited memory space and computation which defers it from running compute-intensive applications. Ad Hoc Cloud System [36] and cuCloud [37] are volunteer systems that harvest resources from sporadically available volunteer nodes, however, they lack locality or performance-aware mechanisms. Some groups have investigated running compute-intensive tasks on edge nodes based on MapReduce [38, 39]. These studies aim to handle resource allocation and data durability, however they are mainly designed for heavy computation with less concern about data storage. In industry, K3s [40] is a lightweight version of kubernetes [41], specifically designed for edge or IoT scenarios. KubeEdge [42] leverage computing resources from the cloud and edge to coordinate both environments. However, they are still oriented to central clusters management without optimization on heterogeneous resources and locality.

Storage at the edge can be categorized into offload (offload data to edge and sync with cloud), aggregate (Data collected from multiple devices to the edge) and P2P (data generated by one device shared with another) [43, 44]. Most of the existing storage systems focuses on offload and aggregate models. P2P storage is not explored much due to concerns of data security and synchronization difficulties across unreliable devices. CloudPath [45] uses PathStore [46], an eventually consistent datastore with persistent data on cloud and partial replicas on edge. The store may have a degraded performance when new data is queried frequently. SessionStore [47] is a hierarchical datastore that guarantees session consistency using session-aware reconciliation algorithms built on top of Cassandra [48] and hence support client mobility to an extend. DataFog [49] is an IoT data management infrastructure which places replica based on spatial locality, addresses sudden surges in demand using a location-aware load balancing policy and evicts and compresses data based on temporal relevance. However, it does not support

network proximity based node selection. FogStore [50] is a geo-distributed key-value infrastructure that places replicas based on latency of data access. Also, to ensure fault tolerance similar to DataFog, one of the replicas is kept at a remote location in FogStore. However, it does not take into account the limited storage capacities of heterogeneous storage nodes.

Chapter 3

Armada System Architecture

3.1 Overview

Figure 3.1 shows the Armada system architecture. Armada consists of geo-distributed nodes that donate their compute and/or storage resources, along with a set of global and central services hosted on dedicated, stable nodes. Both Armada system components and Armada-hosted applications are encapsulated in Docker containers for ease of use and fast deployment. Docker itself provides a lightweight, isolated runtime and abstractions over underlying resources for edge nodes, which is a good option for shipping the code easily to volunteer-based heterogeneous environments. Armada resources and services together constitute the following major components:

- **Beacon:** Beacon is the global entry point for all interactions with Armada central services. It will forward requests to corresponding handler components, including application deployment requests, user connection requests and resource registration requests.
- **Application Manager:** Application manager maintains the states of submitted applications in Armada and manages the application lifecycle. It globally controls, operates, and monitors all application tasks running on different edge nodes, and processes initial user connecting requests. It also handles auto-scaling based on real-time user demand.

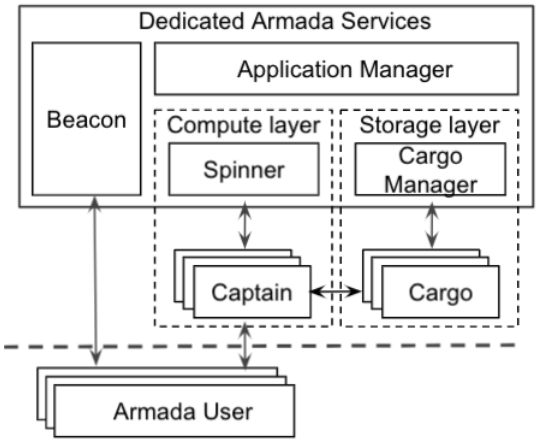

Figure 3.1: Armada system architecture

- **Compute Layer:** Compute layer manages dedicated and volunteer compute resources in Armada. It includes Spinner, the compute resource manager and Captain, the compute node. The Spinner handles compute node registration, health check and resource allocation for task deployment requests sent by the Application manager. The Captain manages the local heterogeneous resources through the Docker engine API and processes user workloads.

- **Storage Layer:** Storage layer manages dedicated and volunteer storage resources in Armada. It includes Cargo manager, the storage resource manager and Cargo, the storage node. The Cargo manager handles storage node registration, health check, maintains metadata and executes storage policies for data-dependent applications. The Cargo manages the local heterogeneous storage resources using the Docker volume and persists data on the edge supporting low-latency access for nearby users.

- **Application Client:** Application client is the user-side program of Armada applications. It incorporates Armada client SDK with application-specific logic to communicate with edge services deployed in Armada.

3.2 Beacon

Beacon is the entry point of contact for all initial interactions with Armada. It exposes interfaces for application developers to deploy edge services and monitor service status, application users to query service access points, and resource contributors to register edge nodes. Requests with different purposes will be forwarded to different handler services i.e., Application manager, Spinner and Cargo Manager, for further processing. Beacon provides the central public access point for different entities to establish initial connections with Armada components.

3.3 Application Manager

Application manager (AM) handles service deployment requests from application developers and service discovery requests from application users. AM also monitors the user demand and user distribution to make service auto-scaling decisions. Each service in Armada contains multiple replicas, namely tasks, deployed on distributed edge nodes. AM globally controls and monitors all replicas of the service through task-oriented APIs exposed by the compute layer (Section 3.4). In this way, Armada decouples the application-level management from the underlying edge resources layer. Three major modules of AM are described as follows.

Parameter	Description
Image	Docker image for the application service
Compute_Req	Compute resource requirements
Sched_Policy*	Optional customized scheduling policy
Location	Coordinate(s) for expected user distribution
Need_Storage	If persistent edge storage is required
Storage_Req*	Storage requirements: capacity, consistency policy and data source

Table 3.1: Service deployment interface(* denotes optional parameters)

3.3.1 Service Deployment

Initial service deployment request includes parameters shown in Table 3.1. Service deployers only need to specify the resources required per replica without worrying about the number of replicas and replica distributions. AM initially deploys a minimum of three replicas to guarantee fault tolerance through the Spinner task deployment API. Then more replicas will be automatically spawned based on actual user demand and distribution (discussed later in auto-scaling). For all deployed tasks, AM periodically requests the underlying resource layer to collect real-time updates including running status, current load and resource utilization. If the *Need_Storage* field is true, AM will send storage resource requirements to the Cargo manager (Section 3.5) to allocate persistent edge storage capacity associated with the service.

3.3.2 Service Discovery and Selection

AM processes service discovery requests for application clients to establish initial connections to edge servers. Multiple factors are incorporated to make service selection decisions including geo-locations, resource availability, server load and network latency. We propose a 2-step performance-aware approach (discussed in Chapter 4) to select the best edge access point for each application client.

3.3.3 Service Auto-scaling

AM handles the auto-scaling of the service based on the real-time user demand and distribution. The initial three service replicas are deployed in expected locations (Table 3.1) without having actual users connected. When users join, AM will asynchronously associate user locations with new task deployment requests sent to the Spinner. Then, the Spinner scheduler will try to incrementally allocate more edge resources in specified locations to deliver better edge performance. With the help of Spinner scheduling policies (Section 3.4), AM auto-scaling requests can adapt to both higher user demand and wider user distribution by deploying more replicas in overloaded locations and spawning replicas in new locations.

In Armada, scalability is achieved at both service deployment and user service selection levels to better allocate edge resources and to balance user workloads, achieving

higher average performance.

3.4 Compute Layer

Armada compute layer manages dedicated and volunteer compute resources to execute latency-sensitive and computation-intensive edge services. It contains Spinner, the compute resource manager, and Captains, the geo-distributed edge compute nodes in Armada.

3.4.1 Spinner

Spinner manages edge compute resources in Armada, and runs the Armada scheduler that allocates edge resources and deploys tasks. Table 3.2 shows Spinner interfaces, including task-oriented APIs for Application mananger to operate on tasks and APIs for Captains to register and report status. Spinner acts as the bridge between Armada applications and underlying edge compute resources.

Interface	Input/Output	Description
Task_Deploy	Task_Metadata/ Status, Task_ID	Application manager sends a task deployment request to Spinner.
Task_Status	Task_ID/ Task_Status	Application manager queries the runtime status of the task.
Task_Cancel	Task_ID/Status	Application manager notifies Spinner to remove a task.
Captain_Join	Node_Metadata/ Status	A new Captain registers itself into the system.
Captain_Update	Captain_Updates/ _	Captain sends heartbeats to Spinner reporting status updates.
New_Policy	Schedule_Policy/ Status	Register a new scheduling policy.

Table 3.2: Spinner interfaces

Spinner handles the *Task_Deploy* request through the Armada scheduler. Given

the task image, resource requirements, target location, and optional custom scheduling policies, Armada scheduler uses a series of node filters followed by sorting policies to select edge nodes in heterogeneous environments effectively. We consider four types of policies in the scheduler: locality-based, resource-aware, docker-aware and customized. Our tech report [51] describes the details of scheduling policies.

Filter policies are used sequentially to remove unqualified Captains, while all sorting policies are used collectively to determine the final sorting order. Each sorting policy is subject to a weight, defined as how significantly this policy affects the latency performance. The weighted score decides the final selected Captain for each *Task_Depoly* request. Note that Spinner also notifies un-selected Captains to prefetch the task images if possible to accelerate future task deployment by reducing the image downloading time.

3.4.2 Captain

Captain[1] is an edge compute node in Armada. It listens to task operation instructions from Spinner, manages container lifecycle locally through Docker engine APIs, and discovers nearby edge storage capacity for data-related tasks using Cargo manager (Section 3.5). Captain isolates Armada runtime from the host environments and exposes edge services for direct connections with nearby users. Captain also reports local resource utilization, task running status and image repository information periodically to Spinner.

3.5 Storage Layer

Armada storage layer maintains dedicated and volunteer storage resources in Armada. It enables edge services and applications to persist data on the edge with low-latency access. Armada storage layer consists of two components: Cargo Manager, the storage resource manager, and Cargos, the geo-distributed storage nodes.

[1] Captain represents both the edge compute node and the controller container running in the node.

3.5.1 Storage Manager

Cargo manager manages edge storage resources in Armada. Table 3.3 shows Cargo manager interfaces: for Cargos to join and report status, Application manager to allocate storage resources, and Captains to discover nearby data access points. Cargo manager also spawns data replicas to guarantee fault-tolerance and low-latency data access for geo-distributed services. Data persistence is achieved on edge with redundant data replicas and flexible data consistency policies. The three main modules of Cargo manager are storage registration, data access point selection and storage auto-scaling. Our tech report [51] describes the details of storage policies and mechanisms.

Interface	Input/Output	Description
Cargo_Join	Cargo_Metadata/ Status	A new Cargo registers itself into the system.
Cargo_Update	Cargo_Updates/ _	Cargo sends heartbeats to Cargo manager reporting status updates.
Store_Register	Storage_Req/ Status	Application manager registers storage capacity for an edge service.
Cargo_Discover	Captain_Info/ Status	Captain queries nearby data access points

Table 3.3: Cargo manager interfaces

3.5.2 Cargo

Cargo is an edge storage node in Armada. It handles data I/O operations and propagation of updates to replicas depending on the type of consistency. Each Cargo node is aware of at most three replica Cargo nodes corresponding to application data. The updates made to one Cargo node are propagated in a cascade manner to all the replicas if more data replicas are spawned to meet the user demands. Table 3.4 describes the Armada storage SDK used by server-side application programs to interact with the storage layer. With Captains locating nearby data access points, Armada storage SDK helps tasks transparently communicate with nearby Cargos.

Function	Input / Output	Description
Init_Cargo	Cargo_App_ Metadata/ Status	Establish connection with a Cargo node
Write	Write_Data/ Write_Status	Write data to the Cargo node
Read	Read_Data/ Read_Status	Read data from Cargo node
Close_Cargo	_/Status	Close connection to Cargo node after use

Table 3.4: Armada storage SDK

3.6 Application Client

Application client is the user-side program of Armada applications. It contains the application-specific logic and uses Armada client SDK to help application users locate the service access points and establish connections. Application client plays an important role in coordinating with Armada system components to achieve latency-sensitive service selection, scalability and fault tolerance. We describe performance probing (in Chapter 4) and multi-connection (in Chapter 5) strategies which are core building blocks inside Armada client SDK, and discuss how they are applied to deliver Armada benefits.

Application developers develop the application client program using Armada client SDK with minimum code modifications. We currently support the gRPC protocol in Golang and around 10 lines of code are added to apply the changes in our experiment applications.

Chapter 4

Performance-Aware Service Selection

4.1 2-Step Approach Service Selection

For availability and scalability purposes, multiple edge service replicas are deployed on multiple edge nodes densely distributed around application clients. How to discover and select nearby edge access points becomes the vital challenge to deliver satisfactory latency performance. We implement a performance-aware service selection approach to accurately evaluate edge server candidates and networking environments for each application client to make the correct service selection decision.

Application manager (3.3) processes initial requests for application client connections. It maintains the metadata and states of all deployed service replicas. Application users need to query AM for nearby access points before establishing direct communication channels. However, the networking performance is nondeterministic in heterogeneous wide-area environments, and different hardware leads to different processing speeds. In addition, non-Armada networking traffic and workloads are unpredictable in practical volunteer environments, which will also cause performance fluctuation at random periods. There are no unified criteria to address all the above heterogeneities and system dynamics at the same time.

We argue that periodic end-to-end latency probing is the only effective way to identify the best-performing edge node in real-time deterministically. In Armada, we propose

a *2-step* approach for application clients to select low-latency service access points accurately. Application Manager implements the first step of this approach by generating the service *candidate list*, and application clients finish the second step by performing the probing tests and making final decisions.

4.1.1 Step1: Server-side Service Selection

Algorithm 1 shows how to generate the service *candidate list* at step 1 using the user information as input. The *candidate list* is a small subset of service replicas that are likely to provide low latency responses for specific users. The considered factors include geo-proximity, resource utilization of the service replica (to detect overload), and the optionally-specified network affiliation between edge nodes and users.

Algorithm 1 Service Selection Step-1

Input: $Loc, NetType, \ldots$
Output: $CandidateList$

```
1: function ServiceSelect(Loc, NetType, ...)
2:     LocalServices ← geoProximitySearch(Loc)
3:     for i ← 1 to LocalServices.len() do
4:         EdgeNetType ← LocalServices[i].NetType
5:         LocLocalServices[i].Score ←
               LocalServices[i].Resources * weight1 +
               netAffiliation(EdgeNetType, NetType) * weight2 +
               ...
6:     end for
7:     CandidateList ← TopNSort(TopN, LocalServices)
8:     return CandidateList
9: end function
```

In *geoProximitySearch()*, we apply GeoHash [52] with less precision to identify a wider-range geographical area, so relatively far-away edge nodes will be evaluated in the same way as closer edge nodes to avoid excluding better-performing options from the *candidate list* in heterogeneous environments. For available local service access points, we use the real-time resource utilization to identify the existing workload. Edge nodes with less existing load tend to be selected first. We also consider network affiliation in WAN to identify edge nodes with better network connections to application clients. Both edge nodes and application clients can optionally specify their network information

like local ISPs as attributes *NetType* to apply the *netAffiliation()* function. Edge nodes with the same *NetType* as application clients tend to be selected first to deliver better network connections. A weighted score is calculated for each nearby service access point (Algorithm 1 line 5), preparing for the sorting step at line 7. Additional factors can be specified with customized weight to increase the accuracy of the selection.

TopN (line 7) is the length of the *candidate list*. Larger *TopN* value leads to higher accuracy but also higher overhead during the performance probing step. We use the *TopN* of 3 to have moderate overhead and enough accuracy.

4.1.2 Step2: Client-side Performance Probing

Performance probing is the second step in the service selection process. Once the *Candidate list* is generated in Step 1 (4.1.1), all edge access points in this list are passed to requesting application clients. Each application client then establishes connections to all candidates for probing tests. The candidate with the lowest end-to-end latency is selected to start offloading the actual workload. More importantly, the *2-step* service selection process is performed periodically and asynchronously in the background to adapt to system dynamics. If the selected node is suddenly overloaded or a closer node joins the system later, application clients can always identify the changes and switch to a better edge node if necessary.

Concurrent probing tests from one application client to multiple candidate edge nodes introduce networking overhead. Each request contains data payload which consumes the bandwidth on both client side and server side. We require the service deployer to upload a dummy data payload for probing processing upon application submission. In this case, probing requests only measure the RTT latency and heterogenous processing time on the edge side excluding the data transfer time caused by limited bandwidth. We argue that comparing networking RTT latency and heterogenous processing time on the edge side is sufficient to identify the best-performing candidate.

Also, asynchronous probing in the background consumes additional CPU circles on the client side. Resource-constrained client-side devices can specify the interval time to limit the probing frequency. Application clients are more responsive to system dynamics with higher probing frequency. For each group of performance probing tests to multiple candidates, a *Grace_Period* is applied to avoid continuous connection switch.

The application client only switches to a better performing candidate B from the current access point A when Latency_B <(Latency_A + Grace_Period).

4.2 Scalability and Load Balancing

Elasticity is a well-known limitation of edge computing since widely-distributed edge sites have much less compute capacity compared to large-scale data centers. This challenge is further exemplified in volunteer environments since heterogeneous volunteer edge nodes are more resource-constrained compared to dedicated edge micro data centers. In Armada, we achieve edge scalability and load balancing through loosely-coupled densely-distributed volunteer edge nodes, which is natively supported by our 2-step service selection approach (4.1)

The performance probing strategy measures the end-to-end latency for application clients to identify the performance of each candidate edge node. If one candidate is overloaded by other clients and introduces high queueing delay as well as resource contention, the probing test for this candidate then indicates a bad performance and therefore other candidates are selected to serve the client. Since the probing test reflects both the networking RTT latency and processing performance, the predominant factor determines the final service selection decision. If a closer edge node for a new application client has lower RTT latency but is overloaded by existing local clients, the degraded processing performance becomes the dominating factor. A far-away but idle candidate can be selected in this case to deliver faster responses.

As a result, load balancing is automatically handled since overload can negatively affect the performance probing results. Latency-driven performance probing balances the load and improves edge scalability.

Chapter 5

Multi-Connection Fault Tolerance

5.1 Challenges: Fault Tolerance in Edge vs. Cloud

Large-scale data centers provide excellent fault tolerance features supported by abundant backup resources. While one node fails unexpectedly, a second available node can immediately take over the requests, and maintains the continuousness of the services. Redundancy-based mechanisms along with dynamic load balancing mechanisms of intelligent cloud gateways guarantee zero down time of the service upon failures.

However, small-scale edge sites are highly constrained on redundant resources, and volunteer environments are full of unreliable and limited single-machine edge sites. Small edge failures can easily lead to unrecoverable situations where all user connections are forced to terminate, causing service downtime. For critical applications like autonomous vehicles, any level of service downtime is unacceptable and might cause severe consequences.

5.2 Redundancy-based multi-connection fault tolerance

Similar to pre-establishing redundant connections for handover in the telecommunication domain, we develop a client-side SDK to maintain multiple connections to different edge servers, preparing for potential edge failures. Benefits from the performance probing step in service selection strategy (4.1), each application client already has communication channels to different edge node candidates. We employ these redundant

performance probing connections to deliver fault tolerance support upon edge failures.

Our client-side SDK can automatically detect connection failures and immediately switch to the next available edge node in the candidate list without any delay to guarantee the consciousness of the service. Since all edge access points in the candidate list are selected by step 1 in the service selection process (4.1) and they are sorted by performance in the probing test, the newly-connected edge node is the second-best option that can also deliver low-latency responses. The failed connection is then discarded in the next performance probing test and a new candidate list is updated to replace the failed edge nodes. Compared to establishing a new connection upon edge failure, Armada client SDK eliminates the time required for service discovery and connection establishment.

Establishing redundant and proactive connections introduces overhead on both client side and server side. We argue that this is an inevitable strategy to use when applications have hard requirements on continuous services and edge resources are highly dynamic and volatile. With higher node churn and failure rate of volunteer resources, the value of *TopN* needs to be larger to guarantee robustness. *TopN* - 1 redundant connections are established in this case to provide redundancy. In order to minimize the number of redundant connections, step 1 in the service selection process (4.1) tends to construct a candidate list with a combination of dedicated and volunteer edge nodes. Dedicated edge nodes may not have the best performance, but they provide reliable alternative service access points when a better-performing volunteer node suddenly fails or leaves the system. We argue that the overhead of multi-connection is acceptable since it provides the necessary mechanical support for both performance probing strategy and fault tolerance.

The immediate connection switch has its own tradeoff regarding the additional requirements for Armada applications. Armada tends to support stateless workload when local cached states and data cannot be immediately transferred to the newly-connected edge node candidate for continuous processing. Based on our observations, this requirement can hold for most latency-sensitive computation offloading workload since the compute resources and power supply is the major concern and bottleneck. The application can either be adjusted to maintain the hard states locally on the client sides or utilize the Armada storage layer to align with the stateless requirement. Armada

storage layer (3.5) persists the data on the edge and supports low-latency data access.

Chapter 6

Real-time Inference on Armada

We implement two real-time inference workloads to evaluate Armada performance. Real-time object detection and face recognition are critical building blocks in commonly used applications like augmented reality, cognitive assistance and security surveillance. They are both computational-intensive and latency-sensitive, which require offloading the computation to powerful servers and obtaining processing results in a timely manner. First, we use an object detection workload to demonstrate the workflow of the Armada computing layer. Second, the face recognition workload [53] showcases the coordination between computing and storage layer when Armada application needs persistent edge storage.

6.1 Real-time Object Detection

Figure 6.1 shows the workflow of real-time object detection in Armada. In the service deployment phase (Figure 6.1 (a)) , service deployers first contact Beacon in step (1) to submit the application along with requirements to Armada. Application manager receives this request in step (2) and initiates three task deployment requests sent to Spinner in step (3). Spinner then calls the Armada scheduler to find available edge nodes and place the tasks in step (4). In the end, the tasks deployment status and service deployment status are updated back to the deployers in step (5) - (8).

In Figure 6.1 (b), When users request the object detection service in Armada, they need to first query the system for service access points in step (1) - (4), and then start

sending the video frames for object detection in step (5). Note that *TopN* number of connections are maintained using the *candidate list* obtained from the service selection process.

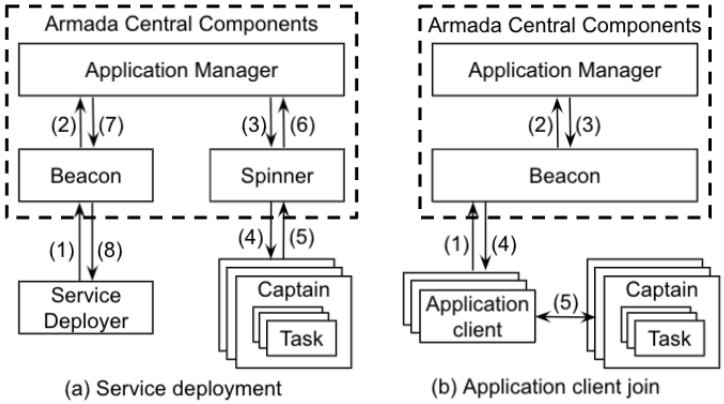

Figure 6.1: Object detection workflow in Armada

6.2 Face Recognition

Figure 6.2 shows the workflow of real-time face recognition in Armada. In the service deployment phase (Figure 6.2(a)), service deployers first submit the application along with requirements for both compute and storage resources (1) - (2). Then the Application manager contacts the Cargo manager to register the storage requirement of the service (3). The Cargo manager selects three Cargos and allocates the required storage resources for three data replicas. The three Cargos then use the specified data source to pull the initial pre-labeled face datasets used to recognize people during the real-time inference. In step (4) - (5), tasks are sent to the compute layer for deployments. To connect tasks with nearby data access points, Captains queries Cargo manager in step (6). Given the *candidate list* of access points, tasks can directly interact with the selected data replicas in step (7) using Armada storage SDK. In the end, the task and service deployment status are updated back to the deployers in step (8) - (11).

Figure 6.2 (b) shows the workflow when face recognition clients request the service. In step (1) - (4), clients first query the system for service access points, and then start sending video frames for face recognition in step (5). For any detected faces during

the processing, tasks query data replicas in Cargos for face recognition (6). The read requests send detected faces to Cargo searching for matched people, and the write requests insert new labeled faces into the persistent data store for future recognition.

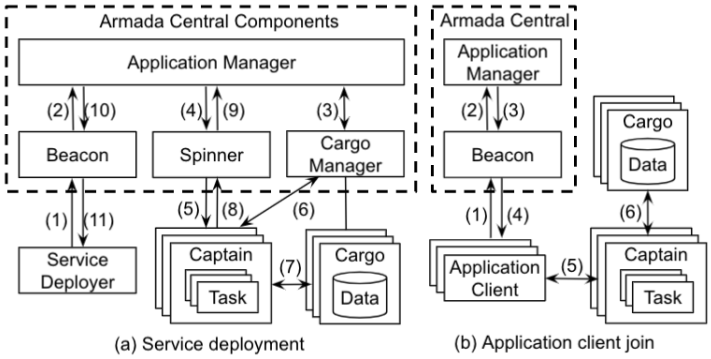

Figure 6.2: Face recognition workflow in Armada

Chapter 7

Evaluation

We evaluate Armada in both real-world edge environments and emulation platforms in the cloud. The real-world experiment explores Armada performance in fine-grained small geographical areas (regions within a city). The emulation experiment explores wider-range geographical areas (regions across nearby cities). We first use a computation-only workload, object detection, to demonstrate Armada service selection, scalability and fault tolerance performance. Then we use a face recognition workload to explore the storage layer performance when the persistent store is required.

7.1 Experimental Setup

In Table 7.1 and Table 7.2, we show the underlying hardware used for both real-world and emulation experiments. Note that the third column shows the processing time per frame for real-time object detection application on these hardwares.

7.1.1 Real-world Environment

We set up the real-world experiment environment around our University campus. As shown in Table 7.1, A combination of both dedicated and volunteer resources is used. Volunteer nodes V1 - V5 are located within 5 miles of the campus, and a powerful University server D6 located on campus is considered the dedicated edge node.

While the dedicated node has more compute power and better network connectivity, volunteer nodes are set up with heterogeneous compute and networking performance

Node	Processor	Processing
V1	Intel® Core™ i7-9700, 8 cores	24ms
V2	Intel® Core™ i7-2720, 6 cores	32ms
V3	Intel® Core™ i9-8950HK, 6 cores	31ms
V4	Intel® Core™ i5-8250U, 4 cores	45ms
V5	Intel® Core™ i5-5250U, 2 cores	49ms
D6	Intel® Xeon® CPU E5-2620 v3, 24 cores	30ms×4
Cloud	t2.large, 4 cores	34ms

Table 7.1: Real-world experiment

Node	Type	Location	Processing
A	t2.2xlarge, 8 cores	City_A	23ms
B	t2.large, 4 cores	City_B	34ms
C	t2.small, 2 cores	City_C	58ms
Cloud	t2.large, 4 cores	Cloud	34ms

Table 7.2: Emulation experiment

contributed by actual volunteers around the campus. The dedicated node D6 can hold four service replicas in parallel, with each of them processing the video at 30ms/frame. Figure 7.1 shows the benefits of exploiting volunteer resources from one user's perspective. Volunteer nodes can deliver similar or even better performance compared to the dedicated edge node.

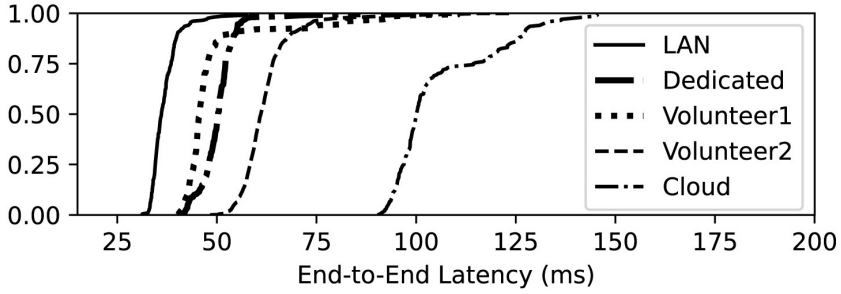

Figure 7.1: CDF of end-to-end latency for different servers

7.1.2 Emulation Environment

Due to physical limits, we use the emulation environment to explore Armada performance on a wider geographical scale. We use the network emulation platform Netropy

[54] in AWS to emulate WAN connectivity for three nearby cities City_A, City_B and City_C that are about 100 - 150 miles away from each other. We configure limited bandwidth on both client and edge sides, and the emulated networking RTT latency is based on the ping tests we perform in real environments. Three edge nodes A, B, C are located at three locations as shown in 7.2.

7.1.3 Baselines

We use geo-proximity, dedicated-edge-only and cloud scenarios for comparisons with Armada.

- **Geo-proximity**: In the geo-proximity scenario, we force all users to connect to the closest edge node in a geographical location, a typical edge selection policy to identify the low-latency edge access point.
- **Dedicated-edge-only**: In the dedicated-edge-only scenario, we assume that only limited dedicated edge resources are available, which is common in today's edge infrastructure deployment. As shown in Table 7.1, we use one powerful dedicated node as compared to 5 resource-constrained volunteer nodes to maintain a reasonable ratio of the availability of dedicated and volunteer resources. We show the benefits of exploiting volunteer resources by comparing them with the dedicated-edge-only scenario.
- **Cloud**: We show the cloud performance as the baseline compared to other scenarios. We use the closest AWS service region US East to deploy the services and assume that the cloud has unlimited scalability with increasing user demand.

7.2 Latency-sensitive Service Selection

We set up three users C1, C2 and C3, in the real-world experiment. They are located around the campus with heterogeneous networking performance to different edge nodes. We also set up three users User_A, User_B and User_C, in the emulation platform and configure them to be at the same locations as nodes A, B and C with corresponding real-world WAN networking performance. Table 7.3 and Table 7.4 show the pairwise

end-to-end latency for object detection application. The bold underlined values refer to the selected service access point in Armada for each user.

Client	V1	V2	V3	V4	V5	D6	Cloud
C1	**38**	47	49	65	72	42	107
C2	43	**35**	56	58	61	45	102
C3	49	50	45	59	71	**42**	112

Table 7.3: End-to-end latency (ms) in real-world environment

Client	A	B	C	Cloud
User_A	**31**	63	89	108
User_B	63	**47**	83	102
User_C	**51**	68	58	111

Table 7.4: End-to-end latency (ms) in emulation environment

The experiment results show that users in both real-world and emulation environments can identify the heterogeneity of the environment and select the best-performing node to offload the workload. In Table 7.4, User_C can select a farther node A due to local resource limitation in node C.

7.3 Scalability and Load Balancing

We explore Armada's scalability performance over high user demand and wide user distribution. We evaluate the average end-to-end latency for the object detection application with a varying number of users and edge nodes.

7.3.1 Performance over increasing user demand

We recruit 15 users around the campus (within 5 miles) with heterogeneous networks to play object detection clients in real-world experiments. With edge resources from five volunteer nodes and one dedicated node shown in Table 7.1, 15 users incrementally start requesting the service. We record the average end-to-end latency at three time slots when there are five, ten and 15 concurrent users. Figure 7.2 shows the user average performance using Armada as well as other baselines.

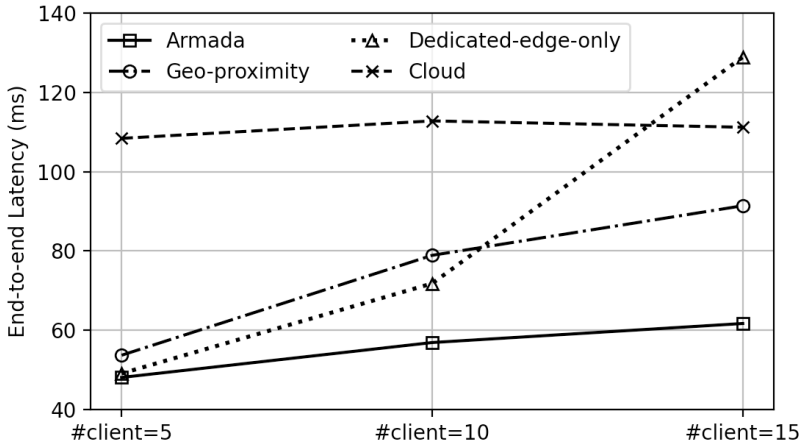

Figure 7.2: Performance over increasing user demand

Armada shows promising scalability performance: 33% faster than the geo-proximity scenario and 52% faster than the dedicated-edge-only scenario at #client = 15 in our experimental setup. First, locality-based service selection ignores network heterogeneity and quickly leads to performance degradation caused by overload. Second, dedicated edge resources are limited in point-of-presence and elasticity. High concurrent user demand can easily overload an edge cite as shown in Figure 7.2, where the dedicated-edge-only scenario is even worse than cloud performance at #client = 15.

7.3.2 Performance over wide user distribution

In this emulation experiment, we explore Armada scalability and load balancing behaviors in wide area settings.

Varying no. of users with a fixed set of edge nodes: In Figure 7.3, with static edge nodes A, B and C as described in **??** (b), we incrementally add users to different cities and observe the average latency performance for users at each city. Each subfigure tells the user distribution and the notation table tells the user edge selection results in Armada. Figure 7.3 (a), as an example, has one user at City_A, one user at City_B and zero user at City_C. The City_A user selects node A and the City_B user selects node B for processing. We also show the latency performance for locality-based edge selection and cloud as comparisons with Armada.

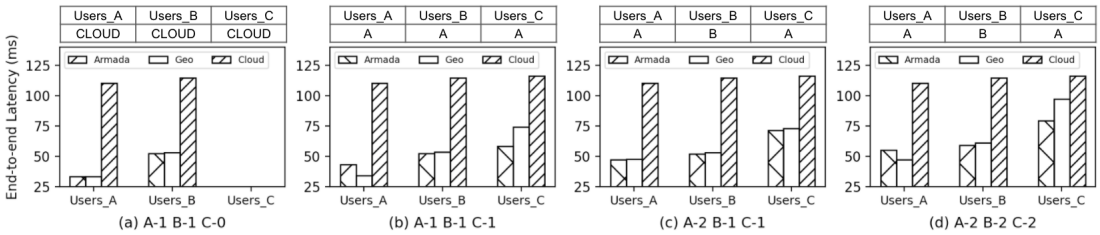

Figure 7.3: End-to-end latency: varying no. of users with fixed set of edge nodes. Each subfigure shows performance under different user distributions. The notation table tells the user edge selection results in Armada.

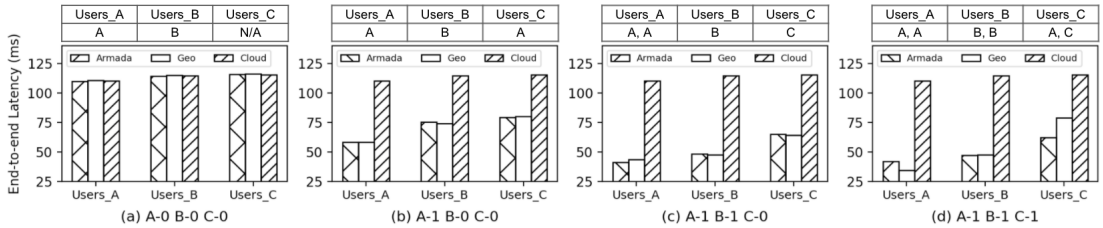

Figure 7.4: End-to-end latency: varying no. of edge nodes with fixed set of users. Each subfigure shows performance under different edge node distributions. The notation table tells the user edge selection results in Armada.

Figure 7.3 (b) shows that the user at City_C selects node A for processing since node A is more powerful and has better performance compared to local node C. Figure 7.3 (c) shows that when two local users are present at City_A, the user at City_C switches back to local node C since node A is fully loaded serving local users. Figure 7.3 (d) shows that when node C is already serving a local user, the second user selects the farther node A after performance probing comparisons. Note that the average performance for users at City_A in Figure 7.3 (b) and (d) are worse than the locality-based approach because local node A serves more users from other cities.

Varying no. of edge nodes with a fixed set of users: In Figure 7.4, with three static users at three cities, we incrementally add edge nodes to observe the user performance. Subfigure captains tell the edge node distribution in this case. Figure 7.4

(b) shows that a new node at City_A improves the performance of all three users in different cities. Figure 7.4 (c) shows that a new node at City_B further improves the performance of all three users. The user at City_B switches to local node B and releases more resources in node A. Figure 7.4 (d) shows that a new node at City_C does not affect the performance because the powerful node A delivers a better performance to the user at City_C.

7.3.3 Fast auto-scaling and Captain registration

We also explore the task deployment speed during the service auto-scaling process. When multiple edge nodes satisfy the task deployment requirements, Armada scheduler tends to select node candidates with overlapping dependencies and base images to accelerate the task deployment process. Armada has unlimited potential to expand with the help of volunteer nodes, therefore, we also explore the Captain registration time and resource usage during idle time to explore Captain lightweight characteristics compared to K3s [40] and K8s [41]. The details of fast auto-scaling and Captain registration performance is discussed in our tech report [51]

7.4 Fault Tolerance

Armada uses the user-driven multi-connection strategy to guarantee continuous service over edge failures. We evaluate the Armada fault tolerance performance in the real-world experiment environment with the object detection workload.

Figure 7.5 (a) shows the end-to-end latency for continuous video frames from a single-user perspective. When the currently connected edge node suddenly fails or leaves the system, the Armada client can immediately switch to a backup node and prevent the service downtime compared to a server re-connect approach.

In Figure 7.5 (b), we manually fail edge nodes one by one and observe the average end-to-end latency of ten static users after each failure. The service is always guaranteed to be continuous in this experiment. So, as comparisons, we develop an Edge-to-Cloud approach where the end-user can immediately switch to the cloud due to node failure. The value on top of each data point (say 8/10) shows the number of still connected

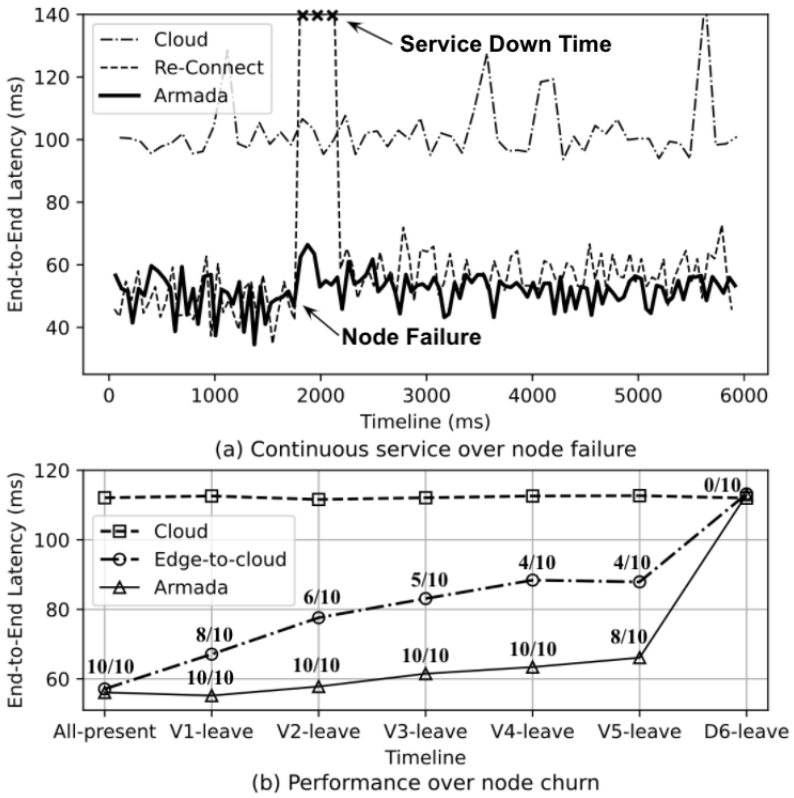

Figure 7.5: End-to-end latency over node churn. The ratios over data points show the number of users that are still connected to edge nodes.

users to the edge after each node failure. With all the edge nodes failing, both Edge-to-Cloud and Armada approaches show cloud performance at the end. However, Armada shows a lower average latency since the failed users switch to alternative edge nodes for low-latency processing.

7.5 Performance of Storage Layer

We use the face recognition workload to evaluate storage layer performance in real-world environments. Since edge nodes in Armada are volatile and dynamic, data persistence poses challenges considering low-latency requirement of data access and storage capacity limitation on the edge. Armada storage layer introduces latency-sensitive Cargo selection mechanisms and flexible data duplication policies with different consistency

levels to support low-latency data access and reliable data storage on the edge.

We evaluate the effects of the Cargo selection strategy, storage fault tolerance and different consistency policies in our tech report [51].

Chapter 8

Conclusion and Future Work

Motivated by powerful and highly-distributed edge resources that are underutilized, we presented the design of Armada, a densely distributed edge cloud infrastructure running on dedicated and volunteer resources. The lightweight Armada architecture and system optimization techniques were described, including performance-aware edge selection, auto-scaling and load balancing on the edge, fault tolerance, and in-situ data access. We illustrated how Armada served geo-distributed users in heterogeneous environments. An evaluation was performed in both real-world volunteer environments and emulated platforms. Compared to the locality-based approach and dedicated-resource-only scenario, Armada shows a 32% - 52% reduction in average end-to-end latency when local users tend to overload the available edge resources. We use a small-scale real-world experiment configuration with 15 users and limited volunteer edge nodes to showcase the scalability feature of Armada. We also plan to develop an experimental platform to verify the scalability feature in a large-scale setting.

A true acceptance of edge computing and emerging edge applications are based on a wide deployment of edge computing infrastructure, which requires tens of thousands edge site Point-of-Presence (PoPs) to be widely distributed around end users and data sources. Today's edge resource availability contributed by major cloud providers and on-premise deployments is far from the expectation level. We argue that an effective edge computing platform in the future should be constructed with loosely-coupled contributors from different companies, organizations, institutions and individuals to guarantee a wide and dense distribution of edge resources. Heterogeneous and volunteer-based

dynamic environments will become main challenges to allocate resources and provision edge workloads.

For future work, an optimization problem needs to be formulated in volunteer edge-dense environments to minimize the average end-to-end latency for application clients through better service selection and placement decisions. We should also consider multiple applications with heterogeneous QoS requirements to improve overall resource utilization. Different QoS requirements of applications and geographically-distributed application clients can guide resource allocation to satisfy the latency requirements of more users. An online churn analysis is required to quantify the volunteer node stability, which plays an essential part in the service placement process. Furthermore, in the Armada storage layer, different data placement, duplication, consistency and migration policies will be explored to dynamically cater the need of latency-sensitive applications.

References

[1] Mahadev Satyanarayanan, Guenter Klas, Marco Silva, and Simone Mangiante. The seminal role of edge-native applications. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 33–40, 2019.

[2] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, Daniel Siewiorek, and Mahadev Satyanarayanan. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, page 45–50, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 152–165, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Amazon. Aws local zones, 2021.

[6] Amazon. Aws wavelength, 2021.

[7] Microsoft. Azure edge zones, 2021.

[8] Google. Global mobile edge cloud, 2021.

[9] Mutable. Mutable, 2021.

[10] EDJX. Edjx, 2021.

[11] MobiledgeX. Mobiledgex, 2021.

[12] Berat Can Şenel, Maxime Mouchet, Justin Cappos, Olivier Fourmaux, Timur Friedman, and Rick McGeer. Edgenet: A multi-tenant and multi-provider edge cloud. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, page 49–54, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Akamai. Akamai cdn, 2021.

[14] Amazon. Amazon elastic compute cloud, 2006.

[15] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[16] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, page 13–16, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[18] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[19] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 68–81, New York, NY, USA, 2014. Association for Computing Machinery.

[20] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 159–173, 2018.

[21] Naser Hossein Motlagh, Miloud Bagaa, and Tarik Taleb. Uav-based iot platform: A crowd surveillance use case. *IEEE Communications Magazine*, 55(2):128–134, 2017.

[22] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[23] Robert-Steve Schmoll, Sreekrishna Pandi, Patrik J. Braun, and Frank H.P. Fitzek. Demonstration of vr / ar offloading to mobile edge cloud for low latency 5g gaming application. In *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–3, 2018.

[24] Seyed Yahya Nikouei, Yu Chen, Sejun Song, Ronghua Xu, Baek-Young Choi, and Timothy R. Faughnan. Real-time human detection as an edge service enabled by a lightweight cnn. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 125–129, 2018.

[25] Seyed Yahya Nikouei, Yu Chen, and Timothy R. Faughnan. Smart surveillance as an edge service for real-time human detection and tracking. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 336–337, 2018.

[26] Nikhil Sreekumar, Abhishek Chandra, and Jon Weissman. Position paper: Towards a robust edge-native storage system. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 285–292. IEEE, 2020.

[27] Alex Glikson, Stefan Nastic, and Schahram Dustdar. Deviceless edge computing: Extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery.

[28] Jianyu Wang, Jianli Pan, and Flavio Esposito. Elastic urban video surveillance system using edge computing. In *Proceedings of the Workshop on Smart Internet of Things*, SmartIoT '17, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Aly Sakr, Seenu Mohiyadeen, Bheeshmaraya Vruksharaj, and Rolf Schuster. Qos-aware score-based edge resource allocation model. In *2020 IEEE 5th International Symposium on Smart and Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS)*, pages 1–7, 2020.

[30] Thomas Lin, Weiyu Zhao, Ivan Co, Andrew Chen, Henry Xu, and Alberto Leon-Garcia. Physarumsm: P2p service discovery and allocation in dynamic edge networks. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 304–312, 2021.

[31] Minh Le, Zheng Song, Young-Woo Kwon, and Eli Tilevich. Reliable and efficient mobile edge computing in highly dynamic and volatile environments. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 113–120, 2017.

[32] David P Anderson. Boinc: a platform for volunteer computing. *Journal of Grid Computing*, 18(1):99–122, 2020.

[33] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *International Workshop on Peer-to-Peer Systems*, pages 205–216. Springer, 2005.

[34] Tessema M Mengistu and Dunren Che. Survey and taxonomy of volunteer computing. *ACM Computing Surveys (CSUR)*, 52(3):1–35, 2019.

[35] Mathew Ryden, Kwangsung Oh, Abhishek Chandra, and Jon Weissman. Nebula: Distributed edge cloud for data intensive computing. In *2014 IEEE International Conference on Cloud Engineering*, pages 57–66. IEEE, 2014.

[36] Gary A McGilvary, Adam Barker, and Malcolm Atkinson. Ad hoc cloud computing. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 1063–1068. IEEE, 2015.

[37] Tessema M Mengistu, Abdulrahman M Alahmadi, Yousef Alsenani, Abdullah Albuali, and Dunren Che. cucloud: Volunteer computing as a service (vcaas) system. In *International Conference on Cloud Computing*, pages 251–264. Springer, 2018.

[38] Kyle Carson, John Thomason, Rich Wolski, Chandra Krintz, and Markus Mock. Mandrake: Implementing durability for edge clouds. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 95–101. IEEE, 2019.

[39] Fernando Costa, Joao Nuno Silva, Luís Veiga, and Paulo Ferreira. Large-scale volunteer computing over the internet. *Journal of Internet Services and Applications*, 3(3):329–346, 2012.

[40] K3s. K3s: Lightweight kubernetes, 2021.

[41] Eric A Brewer. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*, pages 167–167, 2015.

[42] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377. IEEE, 2018.

[43] Enzo Baccarelli, Paola G Vinueza Naranjo, Michele Scarpiniti, Mohammad Shojafar, and Jemal H Abawajy. Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study. *IEEE access*, 5:9882–9910, 2017.

[44] Paola G Vinueza Naranjo, Enzo Baccarelli, and Michele Scarpiniti. Design and energy-efficient resource management of virtualized networked fog architectures for the real-time support of iot applications. *The journal of Supercomputing*, 74(6):2470–2507, 2018.

[45] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework.

In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[46] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. Pathstore, a data storage layer for the edge. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 519–519, 2018.

[47] Seyed Hossein Mortazavi, Mohammad Salehe, Bharath Balasubramanian, Eyal de Lara, and ShankaranarayananPuzhavakathNarayanan. Sessionstore: A session-aware datastore for the edge. In *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pages 59–68. IEEE, 2020.

[48] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[49] Harshit Gupta, Zhuangdi Xu, and Umakishore Ramachandran. Datafog: Towards a holistic data management platform for the iot age at the network edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[50] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. Fogstore: Toward a distributed data store for fog computing. In *2017 IEEE Fog World Congress (FWC)*, pages 1–6. IEEE, 2017.

[51] Lei Huang, Zhiying Liang, Nikhil Sreekumar, Cody Perakslis, Sumanth Kaushik Vishwanath, Abhishek Chandra, and Jon Weissman. Armada: A robust latency-sensitive edge cloud in heterogeneous edge-dense environments, 2021.

[52] Zoran Balkić, Damir Šoštarić, and Goran Horvat. Geohash and uuid identifier for multi-agent systems. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, pages 290–298. Springer, 2012.

[53] Kagami. go-face, 2021.

[54] Apposite Technologies. Netropy emulator, 2021.